MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Computation Structures Group Memo 97

Proposed Research on Semantic Foundations for Structured Programming

Part II of a Proposal to NSF
for Further Research in Computation Structures

by

Jack B. Dennis
Barbara H. Liskov

February 1974

PART II. SEMANTIC FOUNDATIONS FOR STRUCTURED PROGRAMMING

A. SUMMARY

This research program will develop and evaluate fundamental semantic constructs to support the design, construction and verification of well-structured programs. New linguistic features required for structured programming will be identified and defined through the development of a structured programming language. Semantic constructs for realizing these linguistic features will be incorporated into the definition of a base language. A realization of the base language will be designed in the form of an experimental computer system having characteristics matched to the semantic constructs of the base language. The program will include supporting studies in formal semantics, schematology, program verification and computer architecture.

A separate proposal for funding construction of the envisioned experimental computer system will be submitted when our design effort has reached the point that hardware requirements can be specified. We expect this will occur during the first year of the research program.

B. INTRODUCTION

In recent years the cost of producing software has grown to the point where it dominates all other costs involved in making use of computers [1]. Not only is software costly to produce, but when it is delivered it is full of errors, and is difficult to maintain and modify.

Software is expensive because of its complexity [2]. Much of the complexity is in the problems which software is asked to solve -- the problems are inherently complex. However, additional complexity arises from two sources: (1) from our inability to discover the concepts required to solve a complex problem, and (2) from difficulties in representing a problem solution as a program for execution on a computer system. The latter added complexity does not reflect complexity in the problem or in the concepts required for its solution, but rather arises from the mismatch of the semantic constructs provided by the computer system and its programming languages to the representational and transformational needs of the problem. For example, there may not be enough primary memory available, so memory management is added to the problems which software must solve. Or, the means provided for organizing concurrent processing activities is inconsistent with the modular structure of programs.

Progress towards production of understandable and correct software requires techniques for keeping complexity under control -- tools that help the programmer in organizing the inherent complexity of his problem, while adding the fewest extraneous issues to his task. The discipline of structured programming is concerned with the development of such techniques. Interest in structured programming was sparked by Dijkstra's letter [3] in which he argued the existence of a relationship between quality of programs (correctness, understandability) and the absence of goto's and labels. Although the ensuing debate led many programmers to associate structured programming with goto-free programming, Dijkstra himself identifies structured programming with the development of program design methodologies [4], and much interesting work has been done in this area [5,6,2]. This work is closely related to studies of proof of program correctness [7]. Ideally, the programmer should be sufficiently convinced of the correctness of each step in writing a program that he could give a proof of its correctness even though program parts remain to be written. Indeed, the proof of correctness may even precede and define the program [8,9].

In this research on programming methodology, the goto is not the only programming construct to come under criticism. Global variables [10], pointers [11] and mixing of input and output parameters of procedures [12] have been identified as harmful to ease of understanding and verification of programs. Dijkstra has remarked [13] that no present programming language provides good support for structured programming. Thus an important research objective is the design of a programming language that meets the semantic needs of structured programming. We propose to design and specify such a language.

Everyone is aware that the new technologies of hardware production should be used to reduce the cost of software development. However, innovation in computer architecture has, with but few exceptions, responded primarily to the twin demands of compatibility with existing languages and systems, and competitive performance. Much innovation simply uses new technology (e.g. microprogramming) to move functions formerly performed by software into the hardware. The result is then considered a "high-level" machine. Choosing the right software functions for micro-coding should result in better performance, but there is little basis for arguing that "high-level" hardware will, of itself, reduce the cost of software development.

The most successful early example of a high-level machine is the Burroughs B5000 and its successors, which have built-in stacks designed to match the conventions of Algol 60 for accessing variables [14]. Yet the B5000 cannot be considered an "Algol 60 machine" because the hardware includes a complete set of conventional facilities through which many needed programming facilities not encompassed by Algol 60 are provided.

In contrast, the Symbol Project [15,16,17,18] undertook the combined design of a language and its hardware realization -- the language became the functional specification for the machine. The Symbol language frees the programmer from certain restrictions usually imposed by conventional languages to permit efficient implementation on conventional machines: The data of Symbol are tree-like objects that may change in form, depth and extent during execution. (This removal of unnatural constraints from programming languages is certainly in the spirit of reducing complexity of software -- it allows the programmer to represent the entities of his problem more directly by structures of the language.) By using wired-in algorithms for creating and accessing the basic tree-structured objects of the language the Symbol machine realizes the Symbol language with acceptable efficiency.

More recently, many machines have been built (or proposed) to support higher level languages, for example, APL [19] and Lisp [20]. Although such machines can run programs faster than software implementations and perhaps provide better run-time diagnostic facilities, they make little further contribution to reducing the cost of software than is offered by using a high-level language.

It has become evident that computer manufacturers see great potential in moving"established" software functions into microprograms. "Established" software functions may include scheduling and memory management functions, data base access methods, storage mapping functions for structures such as arrays, as well as error recovery and diagnostic features. Since these machines are intended to support several standardized programming languages and provide special support for data bases, the underlying semantic model is at best a kind of ad hoc marriage of "desirable" software features. The outcome promises to be computer hardware that is comparable in complexity of function and specification to present-day operating systems, and, just as for current operating systems, has no easily comprehended semantic model.

Again, this course offers no certainty of reducing software costs. In fact, the software problem will become more difficult as more programs are written that depend on poorly-understood features of the machines.

We believe a different course of development has good potential and must be seriously explored. Experience with higher-level language machines has proved the feasibility of building hardware to support a well-defined semantic model. However, to reduce the cost of software it is necessary to transcend the limitations of conventional languages. We propose to attack the cost of software by identifying new linguistic features helpful in producing well-structured programs. A base language will be developed to serve as a precise semantic foundation for the proposed structured programming language. The model defined by the base language must be sufficiently powerful to encompass the representational needs of a large class of structured programs, yet it must be simple enough that the axioms of the model can be easily understood by programmers and applied to the verification of their programs. Finally a machine architecture will be designed to efficiently realize the model.

In the following two sections we review our current and past research that leads up to the program we wish to pursue during the next period of support. The first section discusses the direction of our thinking about the design of a programming language suitable for structured programming. This discussion introduces the concept of _function clusters_ which provide a new kind of program module to support the use of abstract data types. The second section traces the development of our ideas about computer architecture and semantic models for computer systems. We point out the value of specifying a _base language_ which can serve as the common representation for programs and data structures, and as a functional specification for a computer system. We then discuss a possible form of base language that embodies the notion of data flow rather than control flow. We demonstrate that the data flow language is well matched to the semantic requirements of the proposed structured programming language by showing how it provides a natural representation for function clusters.

The final section describes the proposed program of research.

C. STRUCTURED PROGRAMMING LANGUAGE

We have noted that research in structured programming is concerned with the development of program design methodologies to control the inherent complexity of programs in a manner leading to correct programs which are easy to understand, maintain and modify, and that work in this area is hampered by the lack of a programming language in which structured programs can be written concisely and naturally. During the current period of National Science Foundation support, we have undertaken to design such a language. In this section we first describe the present state of the language. Next we relate the constructs of the language to proofs of correctness. Finally we discuss some problems in the language which are currently being studied.

Our researches in the area of structured programming led us to the realization that a structured program consists of a hierarchy of abstractions [21], and that the ability to identify, use and define abstractions is fundamental to the process of structured programming [2]. Two kinds of abstraction are required: abstract operations and abstract data objects. Existing languages provide support for abstract operations in the form of the function or procedure, permitting an operation to be used abstractly, and its definition to be given separately. A similar separation of use from definition for abstract objects is not provided by existing languages [22]. Our structured programming language will provide this ability through a new linguistic construct supporting the use and definition of abstract data types. An abstract data type is a set of operations. The operations as a whole define the behavior of objects of that type. Users of abstract objects make use of the defining operations, but have no knowledge of how the objects are represented in storage, nor of how the defining operations are implemented. The implementation of an abstract data type, given separately from the use of objects of that type, is provided by a new programming language construct, the function cluster.

C1. USING ABSTRACT DATA TYPES

As an illustration of the use of abstract data types, consider the construction of a translator:

Polish_gen: procedure (input: token_stream, output: outfile) returns outfile;

Procedure Polish_gen maps parameter input, containing a stream of tokens derived from a program expressed in an infix language, into a Polish post-fix language, entering the symbols in parameter output at the appropriate time. For example, if Polish_gen received as input a token stream equivalent of

$$a + b * (c + d),$$

it would return with output containing

$$a\ b\ c\ d + * +$$

The program Polish_gen is shown in Figure 1. It is written in our structured programming language. A preliminary version of the language is described in more detail in Attachment A* and is derived mainly from Pascal [23].

(1) The language has two forms of modules corresponding to the two kinds of abstraction: procedures, which support abstract operations, and clusters which support abstract objects. Polish_gen is an example of a procedure module (an example of a cluster module will be given in the next section). Each module is translated (compiled) by itself.

*B. H. Liskov and S. N. Zilles. An Approach to Abstraction. Draft of a paper accepted for presentation at the Symposium on Very High Level Languages, Santa Monica, March 1974.

```
Polish_gen: procedure (input: token_stream, output: outfile) returns outfile;

 s: stack(token);
 t: token;
 mustscan: Boolean;

 stack$push (s, token_stream$next (input));
 mustscan := true;
 while ¬stack$empty(s) do
    if mustscan
       then t := token_stream$next(input)
       else mustscan := true;
    if token$is_op(t)
       then
          case token$prec_rel(stack$top(s), t) of
             "<":: stack$push(s, t);
             "=":: stack$erasetop(s);
             ">":: begin
                      outfile$out_str(output,
                         token$symbol(stack$pop(s)));
                      mustscan := false;
                   end
          otherwise error;
       else outfile$out_str(output, token$symbol(t));
 end;
 outfile$close(output);
 return output;
end Polish_gen
```

Figure 1

(2) The only free variables which a module con contain are those which identify other modules. These names are bound at translation time by means of a library created by the programmer expressly for this purpose; no free variables remain to be bound in the translated module. Polish_gen makes use of free variables token_stream, outfile, stack and token, all of which identify datatype modules.

(3) The language has only structured control. There are no goto's or labels, but merely variants of concatenation, selection (if, case) and iteration (while) constructions. A structured error-handling mechanism is under development. At present it is represented only by the presence of the reserved word error.

(4) The same syntax is used to declare objects of abstract data type as is used to declare objects of primitive language-supported type. For example

```
s: stack (token)
```

states that s is the name of a variable which holds an object of abstract type stack, and a stack object is to be created and stored in s. Information required for creating the object is passed as a parameter; in the example, parameter token defines the type of element which may be placed on the stack s.

(5) The language is strongly typed. This means that objects may only be operated on by the operations defining their type. Application of a defining operation to an abstract object is indicated by a function call in which a compound name for the function is used: for example

```
stack$push(s, t)
token$is_op(t)
```

The first part of the compound name identifies the type of the operation while the second component identifies the operation.

A brief description of the logic of Polish_gen can now be given. Polish_gen uses variable t to accept tokens from parameter input one at a time. If t holds a token representing a symbol (like a) rather than an operator (like +), that symbol is put in the output file immediately. Otherwise, the token on top of the stack is compared with t to determine the precedence relation between them. If the relation is "<", t is pushed on the stack (e.g. "+" < "*"). If the relation is "=", both t and the top-of-stack token are discarded (e.g. "(" = ")"). If the relation is ">", the symbol held in the top-of-stack token is appended to the output file, exposing a new top-of-stack token. The next comparison will be between the new top-of-stack token and t. All of the token stream has been processed when the stack becomes empty. (We have made the simplifying assumption that the input is a legitimate sentence of the infix language.)

Polish_gen makes use of four data abstractions: token_stream, outfile, token and stack. The power of the data abstractions may be illustrated by considering the type outfile which is used to shield Polish_gen from any physical facts concerning its output. For example, it does not know what output device is being used (or even whether there is one) or when the I/O actually takes place, nor does it know how characters are represented on the device. What it does know about output is just enough for its needs: How to add a string of characters and how to signify that the output is complete. Its knowledge consists of the names of the operations which provide these services.

C2. DEFINING ABSTRACT DATA TYPES

In this section, we describe the programming object, the function cluster, or cluster for short, whose translation provides an implememtation of a type. The cluster embodies the idea of a data type being defined by a set of operations. A cluster exists to support an abstract data type, and each permitted operation corresponds to a function in the cluster.

As an example, consider the abstract data type stack used by Polish_gen. A cluster supporting stacks is shown in Figure 2. This cluster defines a very general kind of stack object in which the type of the stack elements is not known in advance. The cluster parameter element_type indicates the type of element a particular stack object is to contain.

The first part of a cluster definition provides a very brief description of the interface which the cluster presents to its users. The cluster interface defines the name of the cluster, the parameters required to create an instance of the cluster, and a list of the operations defining the type which the cluster implements: e.g.,

stack: cluster (element-type: type) is push, pop, top, erasetop, empty

The use of the reserved word is underlines the idea of a data type being equivalent to a group of operations.

The remainder of the cluster definition, describing how the abstract type is actually supported, contains three pieces of information:

(1) Object Representation. Users of the abstract data type view objects of that type as indivisible, non-decomposable entities. Inside the cluster, however, objects are viewed as decomposable into elements of more primitive type. The rep description defines the way objects are viewed within the cluster, by defining a template which permits

```
stack: cluster(element_type:type) is push,pop,top,erasetop,empty;

   rep(elem_type:type)=(tp:integer;
                        e_type:type;
                        stk:array[1..] of elem_type);

   create

      s: rep(element_type);

         s.tp := 0;
         s.e_type  := element_type;
      return s;
      end

   push: operation(s:rep, v:s.e_type);

      s.tp := s.tp+1;
      s.stk[s.tp] :=v;
      return;
      end

   pop: operation(s:rep) returns s.e_type;

      if s.tp=0 then error;
      s.tp := s.tp-1;
      return s.stk[s.tp+1];
      end
   top: operation(s:rep) returns s.e_type;

      if s.tp = 0 then error;
      return s.stk[s.tp];
      end

   erasetop: operation(s:rep);

      if s.tp=0 then error;
      s.tp := s.tp-1;
      return;
      end

   empty: operation(s:rep) returns Boolean;

      return s.tp=0;
      end

   end stack
```

Figure 2

objects of that type to be built and decomposed. For example, a stack is composed of three subelements named tp, stk, and e_type. The storage for the stack is in the array named stk which contains elements of type e_type, and tp holds the index of the topmost element in the stack.

(2) Object Creation. The reserved word create marks the create_code, the code to be executed when an object of the abstract type is created. The cluster may be viewed as a procedure whose procedure body is the create-code. When a user declares a variable to be of abstract type, for example,

```
s: stack(token)
```

one thing that happens (at execution time) is a call on the cluster procedure, causing the create-code to be executed. The create-code makes use of the cluster parameters which are in fact local to it and may not be accessed in other parts of the cluster-definition (including the rep).

The code shown in the stack cluster is typical of create-code. First, an object of type rep is created: Space is allocated to hold the object as defined by the rep. Then, some initial values are stored in the object. Finally, the object is returned to the caller. As the object is returned, it changes from type rep to the abstract type defined by the cluster.

(3) Operations. The body of the cluster consists of operation definitions, which provide implementations of the permissible operations on the data type. Operation definitions are like ordinary procedure definitions except that they have access to the rep of the cluster,

which permits them to decompose objects of the cluster type. Operations do not constitute modules, but may be compiled only as part of the cluster.

Operations always have at least one parameter -- of type rep. Because the cluster may simultaneously support many objects of its defined type, this parameter tells the operation the particular object on which to operate. Note that the type of this parameter will change from the abstract type to type rep as it is passed between the caller and the operation.

C3. PROOFS OF CORRECTNESS

Structured programming is very much concerned with the construction of programs which can be proved correct. In this section we discuss how proofs of correctness can benefit from the concept of abstract data type.

The correctness proof for a structured program consists of proofs of correctness of each of its component program modules, and a proof that correctness of the modules implies correctness of the program. One finds that the details of these proofs are simplified by the restrictions imposed in structured programming: Avoiding use of goto's allows a program module to be treated as a black box with respect to control; the absence of free variables in a module means that all changes effected by module execution are transmitted through its output parameters -- a proof may disregard the possibility of side effects.

A specification for a program module is information that serves as an interface between the implementer and users of a module: The program using the module is proved correct in terms of the specifications, and correctness of a realization of the module is established by proving that it satisfies the specifications. Thus the specification of a module is exactly that information about the module that is required to prove correctness of any program in which the module is used.

Success of the constructive approach to program correctness depends on one's ability to construct clear specifications for program modules. Such a specification should provide a complete prescription of module behavior, but should be independent of module implementation decisions made by the programmer. Clusters supporting abstract data types are an attractive form of program module because implementation decisions concerning representation of objects, and choice of algorithms for the operations, are shielded from users of a cluster.

We are studying the potential of an approach to specifying abstract data types [24]. A specification consists of type specifications for the operations of the data type, and a set of equations that define their behavior. As an example we shall give a specification for the stack data type: The type specifications are:

$$\begin{aligned}
&\text{push: stack} \times T \to \text{stack}\\
&\text{top: stack} \to T\\
&\text{erasetop: stack} \to \text{stack}\\
&\text{pop: stack} \to T \times \text{stack}\\
&\text{empty: stack} \to \text{Boolean}
\end{aligned}$$

Here the symbol T denotes the data type of the stack elements, and is a free variable of the specification. The domain of stack objects may be regarded as generated by the composition of finitely many applications of push, using arbitrary elements of T, and starting with the object null representing the empty stack. For example, the stack obtained by pushing 2 and then 7 into an empty stack is represented by push(push(null, 2), 7)

The equation section specifies behavior by giving conditions that the operations of the data type must satisfy for stack objects:

top(null) = error

top(push(s, t)) = t

erasetop(null) = error

erasetop(push(s, t)) = s

pop(s) = ⟨top(s), erasetop(s)⟩

empty(null) = true

empty(push(s, t)) = false

Variables s and t denote arbitrary values in the domains of stack objects and stack elements, respectively. The equations define operations top, erasetop and empty throughout their domains because any stack object is either null (the empty stack), or is obtained by applying push to some stack object. The operation push does not appear on the left side of any equation because each use of push generates a new stack object, the behavior of which is defined only in terms of its effect on subsequent uses of operations top, erasetop, and empty. Operation pop is defined directly in terms of top and erasetop, showing that pop may be regarded as nonprimitive.

In this example, a specific set of expressions was chosen to represent the domain of stack objects. In general, the domain of a data type would be represented by a set of equivalence classes of expressions formed by all possible compositions of operations which yield results of the data type being specified. These equivalence classes are defined by the intersection of all equivalence relations for which the equations are valid. The equations are valid if every valid substitution of expressions for variables in an equation produces left-side and right-side expressions that are equivalent in the relation. A substitution is valid if, for each variable, every occurrence of the variable is replaced by the same or equivalent expressions.

C4. UNRESOLVED LANGUAGE ISSUES

Many issues remain to be decided before the definition of the structured programming language is complete. In this section we discuss briefly two of the most interesting issues. As might be expected, both issues are concerned with abstract data types.

Data types as parameters. It is our intention that types be legitimate values in the language, and as such they can be passed as parameters, both to procedures, and to clusters. When a data type is passed as a parameter to a cluster, the object which that cluster produces (via its create-code) is of a composite type defined both by the abstract type which the cluster supports and the type represented by the parameter. For example, the object produced by the stack cluster for Polish_gen is really of composite type "stack of tokens". We do not fully understand all the implications of user-defined composite types. It is interesting, however, that composite types have existed in programming languages for a long time, since array definitions permit (insist) that the type of the array elements be specified; we hope, by studying composite types, to shed some light on this area of programming language semantics.

Data Types Versus Clusters. It is important to realize that a data type is not the same thing as a cluster. An abstract data type is a concept whose meaning is captured in a set of specifications, while a cluster provides an implementation of a data type. For example, the data type stack is defined by the specifications given above, while the stack cluster in Figure 2 implements the type. It is easy to conceive of other clusters which also implement the data type stack; these clusters would all provide definitions of the five stack operations, but could make use of a different representation for stacks.

The distinction between types and clusters is important because of its impact on modification of programs. The difficulty of making a proposed modification should be measured by the number of type specifications which must be changed. A modification involving only the redefinition of a cluster is very simple because it will affect no other part of the system. Such a modification can have a significant impact on system performance because two clusters implementing the same data type may differ widely in performance (for example, the implementation of a symbol table involving linear search vs. hash coded look-up).

The above discussion implies that we would like to treat type specifications as fixed, but allow variability in the mapping from an abstract data type to the cluster which implements it. This implies that a program using an abstract data type should be bound to the type instead of a parituclar cluster implementing that type. A partial solution to this problem is provided by the mode mechanisms of Aleph-1 [25].

D. COMPUTER ARCHITECTURE AND SEMANTIC MODELS

In this section we review our past work on computer architecture and semantic models. We then introduce a data flow procedure language which is currently the furthest developed semantic model under consideration as the semantic foundation for the proposed structured programming language. We show how this language meets semantic requirements for structured programming by illustrating how the stack cluster discussed above would be translated into a data flow procedure.

D1. RESEARCH IN COMPUTER ARCHITECTURE

A major part of our past research has concerned the abstraction and analysis of issues arising in the design of general-purpose computer systems and, in particular, studies of how new architectural concepts can help in making computer systems more useful to their user communities.

In connection with the conception of Multics [26], we recognized [27] the importance of addressing schemes that provide location-independent access to all information held online on behalf of system users, and the importance of sharing subsystems and data bases among users. In attempting to extrapolate the ideas in Multics to future computer systems we realized that the segmentation and protection mechanisms of Multics, even though the most advanced yet implemented, were conceived from a "computer systems" viewpoint, and met only the most superficial difficulties in constructing modular programs, shared subsystems and data bases. A major further advance in computer architecture would require a thorough and deep study of data and program structure and the development of matched concepts of computer architecture.

Thus the subject of our research changed from the design problems of multiprogrammed computer systems to fundamental semantic constructs that would form a rational basis for the design of computer systems.

A major goal of our work has been to identify system characteristics essential to modular programming [28] -- the ability to use any program written for execution by a computer system as a module in the construction of larger program modules. At the time the conceptual foundation for Multics was developed, we recognized that one essential requirement for modular programming is location-independent access to information such as is provided by the virtual memory concept. Later, after adopting the goal of identifying fundamental semantic constructs, we realized that another requirement is that all program modules be constructed using the same basic notions of data structure, and that conventional location-addressed memory structures do not provide a suitable basis for achieving the goals of modular programming.

This reasoning led to two major decisions regarding the direction of our research:

First, we decided that any proposal for computer system architecture we developed will be according to a functional specification in the form of a definition of the <u>base language</u> to be realized by the computer system -- the base language will be a precise semantic model for the behavior of the computer system. The base language will be designed so desired functions and qualities can be provided to system users, and architectural concepts will be developed to satisfy system objectives through an effective implementation of the base language.

The design of a base language for a general-purpose computer system raises issues often ignored by language designers:

a. Sharing of data and procedures by programs operating on behalf of different system users.

b. Access control.

c. Cooperative multiprocessing.

We have studied these issues by developing precise models for different aspects of the behavior of general-purpose computer systems. For example, Vanderbilt [29] carried out a study of access control, and a study nearing completion by Henderson [30] concerns a class of computer systems in which a very general form of modular program construction is achieved.

The second major decision concerned the choice of data structures of the base language and associated primitive operations. A most important idea in computation is the building of composite data structures from arbitrary component objects, and the selection of component objects from a composite structure. This idea is supported naturally by regarding structured values as trees: each node of a tree is a _record_ or _bundle_ having finitely many component values identified by _selectors_ which are integers or character strings. Each component of a record or bundle either is _null_, is an elementary value (such as an integer, string, truth value, etc.), or is a tree. The basic operations are the _construction_ of a record from its components and _selection_ of a component from a record. For example

```
cons('top': 0, 'elements': x)
```

constructs a record consisting of a component named top which is the elementary value 0, and a component named elements, which is the value of x; the expression

```
z.'elements'
```

selects from record z the component named elements. We have adopted trees as the fundamental structured values for our research on the semantic constructs of a base language.

Our first conception of a computer architecture founded on a base language was presented at IFIP Congress 68 [31]. This proposal, although incomplete in many respects, embodied a number of ideas and conclusions that remain current in our further study of computer system architecture. In particular, we gave the following argument that highly parallel execution is important to the efficient operation of large modular programs.

We assume computer systems will have several physical memory levels representing different compromises between access time and capacity, and that the distribution of data structures among memory levels is managed by the system. (Independently written program modules cannot perform the storage management function, for each module would need knowledge of the storage requirements of the others to make good decisions.) If the meaningful stored values are trees, only a small unit of data can be moved between memory levels for each access request not satisfied at a more accessible level. Otherwise much data would be moved with little likelihood of being referenced. Thus to achieve a high information transfer rate between memory levels, a computer system using trees as basic values must be designed to handle large numbers of concurrent memory transfers. Since concurrent memory requests can arise only from concurrently active program parts, we conclude that many such parts must be present in the machine simultaneously.

This argument is a principal reason for our interest in a base language in which many program parts are identified for concurrent execution. In particular, we have worked toward specification of a data flow language that encompasses all fundamental constructs of programming and thus will be a candidate base language for guiding the design of computer systems. A data flow language exhibits much of the inherent concurrency in programs because two program parts are required to be executed in a particular order only if one part produces a result used by the second. A data flow language was partially described in the IFIP 68 paper [31], and was used to outline an unusual computer organization which would directly implement the data flow language. However, many questions about representing important programming constructs in data flow form were unanswered at that time, and further refinement of this architectural concept was deferred in favor of developing a more complete semantic model.

D2. DEVELOPMENT OF SEMANTIC MODELS

The first explicit structure of a base language [32] was formulated for an exposition of the premises and objectives of our research, and as a vehicle for studying the problems of translating source language constructs, especially those found in advanced block-structued languages. Conventional flow of control through programs was assumed for this exposition because our knowledge of data flow representations had not advanced sufficiently at that time. The base language was defined in terms of an informally described interpreter whose states represent all information that could affect the logical progress of computation being carried out in the system.

Programs in this first form of base language are representations of procedures which accept arbitrary trees as arguments and produce trees as results. Procedures have no external or free identifiers; hence no side effects can result from procedure application -- each value accessed by a procedure activation must be either part of the tree passed to the procedure as its argument, or a component (a data structure or another procedure) of the procedure itself. Thus a procedure (together with its component procedures and data structures) defines a functional mapping of trees into trees. These are desirable characteristics for modular programming.

Part of our work during the current period of National Science Foundation support has concerned use of proposed base languages as targets for translation of conventional programming constructs. Dennis and Amerasinghe [32, 33] have devised rules for translating from a model block-structured language into a base language. The source language permits procedure variables but only simple data values. The target language is an extended version of the base language in [32]. Using a contour model [34] definition for the block-structured language, Amerasinghe has demonstrated correctness of the translation rules. The means used to implement nonlocal references in block-structured programs is to build, for each procedure application, an "external" record having as its components all nonlocal variables required by the procedure, and to pass this record to the procedure as part of its argument structure. This exercise verified the ability of the base language to support all essential semantic constructs of a block-structured language.

A study of translation rules for languages making use of the ref construct (as in Algol 68) and the notion of cells will be completed by Ellis [35] in the near future. In addition, Hawryszkiewycz has shown [36] how the semantics of relational data base systems can be modelled using a variant of the base language in [32].

The choice of a semantic model/base language for use in implementing the proposed structured programming language is not yet definite -- several alternatives are being actively considered, including a semantic model being developed by Henderson [30]. However, our work during the past year has led to new knowledge about data flow representation of programs, and a data flow base language now appears to be a very attractive foundation for structured programming. A current paper, included as Attachment B* to this proposal, describes a data flow language that incorporates our present knowledge about using data flow concepts to model fundamental semantic constructs of programming. A program in this language is a directed graph having two kinds of nodes -- links and actors. Some of the links are input links which receive values (elementary values or trees), and some are output links which deliver values when computation by the program is finished. The formation rules for data flow programs directly correspond to the control constructs advocated for use in structured programming -- composition, selection of alternatives, iteration, and procedures; there is no construct corresponding to a goto, and values delivered at the output links depend only on the values received at the input links.

A data flow procedure is a data flow program having a single input link at which an argument structure (a tree value) is presented, and a single output link at which a result structure (a new tree value) appears when procedure execution terminates. Any data flow procedure defines a functional dependence of result structures on argument structures -- conflicts among the actors that would produce nondeterminate behavior are prohibited by the rules of construction, and no side effects are possible. Values constructed during execution of a data flow procedure are trees and are never altered; new values (trees) are formed using components of existing values; values are discarded when they are no longer required by a computation.

*J. B. Dennis. First Version of a Data Flow Procedure Language. Draft of a paper accepted for presentation at the Symposium on Programming, Paris, April 1974.

The language as described in the Attachment is sufficiently complete and powerful that Dijkstra's eight queens program, for example, is easily translated into a data flow procedure.

D3. REPRESENTATION OF THE STACK CLUSTER AS A DATA FLOW PROCEDURE

To show how the data flow language can be used as a foundation for the proposed structured programming language, a translation of the stack cluster given in Section C into the data flow base language will be discussed. For this purpose, we use a textual language that is readily translated into the data flow procedure language. The body of a procedure is a while program -- a sequence containing assignment statements such as

```
z := x + y
```

conditional statements

```
if ⟨Boolean⟩ then ⟨while program 1⟩
              else ⟨while program 2⟩
```

iteration statements

```
while ⟨Boolean⟩ do ⟨while program⟩ end
```

and procedure applications

```
z := apply(p, x)
```

The input identifiers of a while program are those identifiers that are used before they are assigned to; the output identifiers are those identifiers assigned to within the while program and subsequently used outside.

While programs are easily translated into data flow programs using procedures describe by Fosseen [37]. Now consider extended while programs in which statements that construct records, or select components of records are permitted. Since values (trees) are never altered during execution of extended while programs, the same translation procedures can be used for translating extended while programs into data flow programs.

A procedure is represented as follows:

```
p: procedure;
   ⟨while program⟩
   end p;
```

The while program which is the body of procedure p has a single input identifier arg and a single output identifier res. Within the text in which procedure p appears, p will have as its value the procedure described by the while program. This value may be used in constructing records, and may be used in an apply statement of the form

```
z := apply(p, x)
```

This statement initiates an activation of procedure p with arg bound to the value of x. When the activation terminates the result structure (the value of res) is returned.

To correctly implement a cluster, we must choose a representation for instances of the data type of the cluster (which were called objects in Section C), and specify how statements calling for use of a cluster operation are to be translated. Furthermore, the implementation must only allow an operation to be used with instances of a data type if the operation belongs to the defining cluster.

Instances of a data type will be represented by a special kind of value, called an object, which we add to the base language. Figure 3 shows an object that represents a stack, i.e. an instance of the stack cluster discussed in

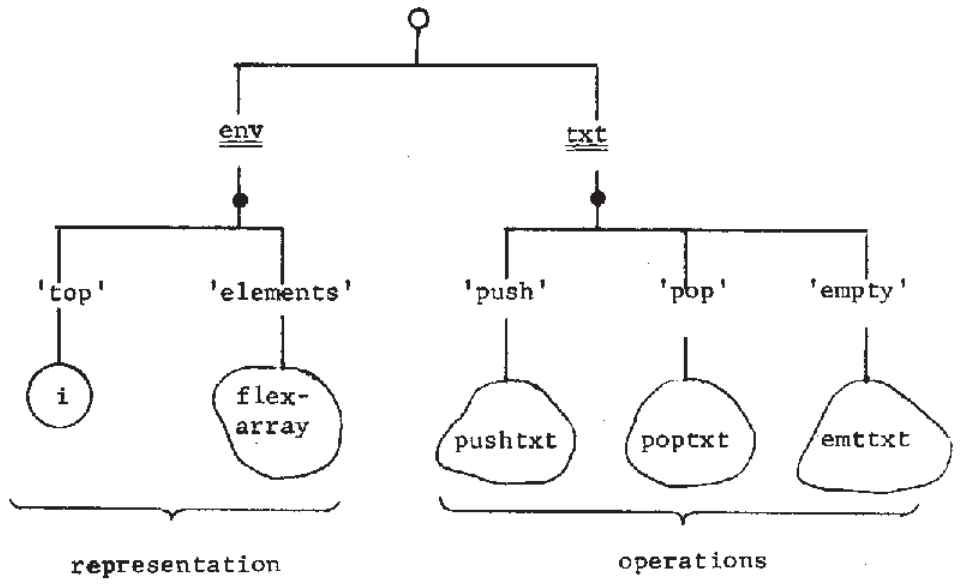

Figure 3. Representation of an instance of the stack cluster.

Section C. An object has two parts identified by the special selectors env (for environment) and txt (for text). For instances of the stack cluster, the env-component contains the representation of a stack and is, for example, a record having an integer component for the stack index and a component of type flex-array for the stack elements. For simplicity the parameter element_type of rep in Figure 2 has been ignored. The env-component is distinct for each instance of a cluster. The txt-component is a record containing one operator for each operation of the data type; for simplicity we consider just push, pop and empty. The txt-component is shared by all instances of a cluster. An object is constructed by the expression object(rep, txt) where rep is any value and txt is a record of operators.

An operator is a new textual construct of the base language, and is an extension of the procedure construct:

```
p: operator;
   ⟨while program⟩
   end p;
```

In the case of an operator, the while program has two input identifiers, env and arg, and two output identifiers, new and res. Finally, we add a new primitive oper which bears the same relation to operators as apply does to procedures. A statement

y := oper(x, 'p', v)

is valid only if x is an object and the txt-component of x contains an operator named p. If so, the operator p is to be applied to object x:

1. Bind the input identifiers of operator p: bind env to the env-component of x; bind arg to v.
2. Execute the body of operator p.
3. Form a new object with new as its env-component and x.txt as its txt-component; make this new object the value of x.
4. Set y = res.

The coding of the stack cluster in this extended language is given in Figure 4. The text as a whole is a procedure containing declarations of operators pushtxt, poptxt and emttxt which implement the stack operations. The procedure constructs and returns an object that represents a stack of zero elements. The data type flex-array is assumed to be implemented as a primitive cluster of the language; a call on procedure flex-array returns a vector of zero elements. In the coding of the operators, certain conventions have been used: If an operation (such as push) produces no result value, no assignment to res is required; if an operator (such as empty) causes no change in the cluster instance, no assignment to new is required. Two operations of the flex-array cluster are used:

```
store(cons('index' = i, 'value' = v))
```

which stores v as the i^{th} element of the array, and

```
v = access(i)
```

which returns the value of the i^{th} element.

The statement

```
stack$push(s, t)
```

in procedure Polish_gen would be translated as

```
oper(s, 'push', t)
```

We see that if oper is the only means for accessing components of objects, then it is impossible to invoke cluster operations except for application to instances of the associated data type. This fulfills one aspect of the strong type checking desired in the structured programming language. However, in

```
stack$push(s, t)
```

the check that s is an instance of the stack cluster is not included in the implementation of Figure 4. Thus s could be an instance of some other cluster having an operation named push. There would be no type conflict between operator and operand, yet the action would not be the programmer's intent. Also, we have included no check that elements of stacks are of a specified type. Means for implementing these checks are being studied.

```
stack:  procedure;

        elmts := apply(flex-array, null);
        value := cons('top' = 0, 'elements' = elmts);
        cluster := cons('push' = pushtxt, 'pop' = poptxt, 'empty' = emttxt);
        res := object(value, cluster);

        pushtext:  operator;
                   s := env; v := arg;
                   i := s · 'top' + 1;
                   elmts := s · 'elements';
                   oper(elmts, 'store', cons('index' = i, 'value' = v);
                   new := cons('top' = i, 'elements' = elmts);
                   end pushtxt;

        poptxt:    operator;
                   s := env;
                   i := s · 'top';
                   elmts := s · 'elements';
                   if i = 0 then error;
                   res := oper(elmts, 'access', i)
                   i := i - 1;
                   new := cons('top' = i, 'elements' = elmts);
                   end poptxt;

        emttxt:    operator;
                   s := env;
                   res := (s · 'top' = 0);
                   end emttxt;

 end stack;
```

Figure 4. Coding for the stack cluster.

D4. ISSUES FOR FURTHER STUDY

Although a large class of programs can be translated into the data flow language, several aspects of programming are not encompassed by the language as presented in Attachment B.

One such aspect is the representation of computations performed by module activations that communicate with each other through _ports_ or _communication variables_. Such computations are often implemented using the techniques of coroutines, but this seems too undisciplined, and has the drawback that the concurrency in the computation is not easily recognized. We have suggested a possible textual form for cooperating modules [38], but the general properties, construction rules and semantics are yet to be worked out in terms of a data flow model.

Another important aspect concerns programs that cannot be represented as modules with functional behavior -- for example, a program that gives several users ability to independently update a file. We plan to design an extension of the data flow language that will encourage the writing of well-structured programs for these applications, while giving the programmer a guarantee of functionality if he avoids use of the primitive constructs that define the extension.

The data flow procedure language incorporates a number of unusual constraints:

1. Absence of _goto_'s
2. All procedures are functional -- no side effects.
3. Values are trees -- cyclic structures are not encompassed.
4. Values are not altered.

These assumptions yield a language in which concurrently executable parts are easily recognized, and which seems attractive for realization on a machine of matched architecture.

We plan to investigate the constraints incorporated into the data flow language and evaluate their significance both with respect to the use of the language as a semantic basis for structured programming and as a new framework for computer architecture.

E. PROPOSED RESEARCH

J. B. Dennis and B. H. Liskov propose to:

1. Continue development of a structured programming language.
2. Design a base language that supports the semantics of the structured programming language.
3. Build an interpreter for the base language and a translator to translate from the structured programming language to the base language.
4. Design and construct a machine that realizes the base language.
5. Build a programming system around the structured programming language to permit convenient use of the language.

It is our intent that the semantics of the structured programming language and the base language be closely related so simple translation rules will suffice for converting structured programs into base language programs. The difference between the two languages is that the base language is an internal representation for procedures and data structures, and its design is concerned exclusively with the cleanest and simplest schemes for representing the desired semantics in an effectively realizable form. The structured programming language, being an external language, must have a textual form which is carefully human engineered for convenient use by programmers. The design of both languages will involve a thorough understanding of the semantics of data types, and we will continue our studies in this area.

As a first test of the structured programming language and base language, we propose to build a software implementation. The implementation will consist of an interpreter for the base language and a translator from the structured programming language to the base language. The process of building these programs will provide a verification of the semantic definitions of both languages. Also, although we expect the implementation to run very slowly, it will enable us to actually execute a few sample programs written in the structured programming language. This exercise will provide valuable information about the human engineering of the structured programming language and about the completeness of the base language.

The machine language of the proposed computer is intended to be simply a transliteration of the base language. The complexity of the machine will be bounded in two ways: We will use the most direct means possible for realizing base language constructs. In this way simplicity of the base language will be reflected in simplicity of the machine architecture. We will also use the most straightforward approach to building the machine. At present we envision assembling the machine from commercial units including microprogrammable processors, packaged memory units and standard peripherals. In this way hardware development effort will be kept low.

A programming system is required if the structured programming language is to see significant use. The programming system will provide filing, editing and debugging facilities designed expressly for use with the structured programming language.

Structured programming is concerned with the efficient development of large programs so as to give confidence in their correctness. Thus the validation of any proposal for aiding the practice of structured programming must include a test of its value in the construction of large programs. A convincing demonstration of our structured programming language will be possible only by making the language available to an appropriate user community for use in building real programs -- an effective implementation of the language is essential.

An adequate implementation of the structured programming language must provide strong data typing of user defined types, storage allocation with retention, and objects whose natural representation is unbounded in both depth and extent. Such constructs are so poorly supported by existing hardware, that a software implementation on a conventional computer will not effectively support the development and use of large, complex programs. The simplest way to provide an effective implementation is to make use of the machine specified by the base language, a machine specially organized to efficiently support the semantic constructs of the language.

Implementation of the structured programming language in terms of a precisely defined base language will also ensure that the semantics of the structured programming language are precise, and provide a basis for proofs of correctness of programs in the language. In addition, such an implementation will serve to evaluate the adequacy and completeness of the base language. If a data flow form of base language is used, we will be able to measure the level of concurrency that may be exploited in the execution of real programs.

Professors Dennis and Liskov are unusually qualified by experience to undertake the proposed project. Professor Dennis wrote and checked out a widely used program for the transportation problem [39], developed hardware and software for one of the earliest time-shared computers [40], and participated in the conceptualization of Multics. Professor Liskov conceived and implemented the Venus system [21], in which the concepts of structured programming were applied to a multi-user computer system constructed from both software and microprograms, and has contributed to the areas of structured programming and design methodology [7, 2].

E1. SUPPORTING STUDIES

In addition to the proposed language and system design project, we expect to continue work in closely allied theoretical areas. These include formal semantics, modes of data, schematology, program correctness, and advanced computer architecture.

Formal semantics: It will be necessary to develop precise definitions of the structured programming language and the base language. This is required because the relationship between the two languages must be thoroughly understood and because the base language is intended to be a precise foundation for the verification of program correctness. Furthermore, since the base language will be the specification for the proposed machine, completeness and accuracy of the definition is important to avoid discovery of ambiguities and contradictions during construction of the system.

It is likely that we will use some form of interpretive model as an aid in assuring correctness of the hardware realization of the base language. However, specifying the semantics of a language axiomatically, as Hoare and

Wirth have done for Pascal [41] is a good way of uncovering poorly designed aspects of the language, and at the same time, stating the semantics in a form directly applicable to proofs of program correctness. An axiomatic definition is particularly natural for the data flow language since all data values that affect the action of a well formed part of a data flow program are passed through its input links, and all values resulting from its action are delivered at the output links -- no alias problems or side effects are possible.

Modes of data: A. E. Fischer and M. J. Fischer have been investigating the general problem of specifying and using properties and relations among modes of data. A mode in their system, Aleph-1 [25], may model an abstract data type or a function cluster, depending on its properties. A particular data object may belong to more than one mode, and a function may be defined on more than one mode. For example, SQUARE_MATRIX can be defined to be a submode of MATRIX which consists of those matrices whose two dimensions are equal. A function such as matrix inverse defined on the mode SQUARE_MATRIX will be applicable only to square matrices, whereas any function defined on MATRIX may be applied to either square or non-square matrices. M. J. Fischer plans to continue this work with A. E. Fischer on the development of Aleph-1 and to validate the design through an experimental implementation, already begun on the PDP-10 at Project MAC. Ways of incorporating their ideas into the structured programming language will also be studied.

Schematology: An important role for the developing theory of program schemas is as a guide to identifying fundamental semantic constructs -- for instance, showing in what sense recursion adds expressive power beyond that possible using iteration [42]. Our work in schematology has been concerned with determinacy of schemas in which concurrency of program parts is represented, and with identifying "nonproductive" parts of program schemas [43]. We have studied formally schemas that model the structure of data flow programs [44], and have found the data flow model to be a useful vehicle for investigating equivalence questions about schemas.

Architecture: We are studying architectural issues raised by assuming a base language as the specification of system function. In particular, we are interested in structures for memory hierarchies designed to store values in the form of trees, and organizations of processing hardware that will yield highly parallel execution of programs expressed in a data flow representation.

E2. SYNTACTIC SPECIFICATION OF PROGRAMMING LANGUAGES

The now classic approach to systematic compiler writing assumes that the language to be compiled is specified using Backus Normal Form (BNF). Many aspects of programming languages are ideally specified in this way.

However, considerable ingenuity may be required to write an efficient compiler for the specified language. The trend is to incorporate this ingenuity into a compiler-compiler in the form of a large quantity of code. The language designer wishing to take advantage of the ingenuity embedded in a compiler-compiler is faced with two unattractive alternatives: He can attempt to transfer an existing compiler-compiler to his computer system, but this is difficult given the current state of the art in softw re portability. Otherwise, he must reimplement the compiler-compiler, which defeats the original goal of easy and convenient compiler construction.

Another problem is that BNF is not ideal for describing every aspect of programming languages. While the canonical example of this limitation is the inability of BNF to describe symbol tables, many trivial but useful constructs are, at best, awkward to describe.

Yet another issue is modularity. While BNF is sometimes advertised as the medium for a structured programming approach to compiler writing, it is possible to write very obscure BNF definitions, with complex interactions between non-terminals. Moreover, nothing about BNF encourages a style of definition that avoids these complexities.

V. R. Pratt has been exploring alternatives to Backus Normal Form and his paper on a top-down operator precedence metalanguage [45] describes such an alternative. The metalanguage differs from BNF in two respects. First, conceptual emphasis is placed on the terminals (lexical tokens) of the language, and second, most of the syntax is implicit in the semantics. The lexical emphasis allows more modularity in a language definition than is generally associated with BNF. This makes a given language easier to implement, provides a simple extension mechanism, and is suited to a compiler which can be partitioned according to the different problem domains of its users. The embedding of syntax in semantics, while not inherently beneficial, in practice turns out to simplify the language definition.

Fischer and Pratt have also considered the problem of implementing the metalanguage. They combine a top-down approach with operator precedence techniques to yield a very fast single-pass parser that is simple to implement. The metalanguage is designed to be interpreted directly, rather than being compiled as is normally the case with BNF, which reduces the size of the system needed for implementing compilers to a trivial amount of code.

These techniques have been used by two groups, the SCRATCHPAD project [46] at IBM, Yorktown Heights, and the MATHLAB project at Project MAC, MIT. Both groups had been using other parsing techniques, but found Pratt's techniques more attractive. In addition, Fischer and Pratt have written a surprisingly compact compiler-compiler that has been used to implement an interactive extensible programming language called CGOL (Computational Generalized Operator Language). One application for this language is as a syntactic front-end for LISP. A small amount of further work is needed on CGOL to make it more accessible to the LISP community of MIT. In addition, the possibility of using this method for the syntactic specification of the structured programming language described above will be investigated.

REFERENCES

1. B. W. Boehm, Software and its impact : a qualitative assessment. Datamation, Vol. 19, No. 5 (May 1973), pp 48-59.

2. B. H. Liskov, A design methodology for reliable software systems. AFIPS Conference Proceedings, Vol. 41, Part 1 (December 1972), pp 191-199.

3. E. W. Dijkstra, Go to statement considered harmful. Comm. of the ACM, Vol. 11, No. 3 (March 1968), pp 147-148.

4. E. W. Dijkstra, Notes on structured programming. Structured Programming (C.A.R. Hoare, Ed.), Academic Press, New York and London, 1972, pp 1-82.

5. D. L. Parnas, Information distribution aspects of design methodology. Proceedings of the IFIP Congress, August 1971.

6. D. L. Parnas, On the criteria to be used in decomposing systems into modules. Comm. of the ACM, Vol. 15, No. 12 (December 1972), pp 1053-1058.

7. B. H. Liskov and E. Towster, The Proof of Correctness Approach to Reliable Systems. Report MTR-2073, The Mitre Corp., Bedford, Mass., July 1971.

8. E. W. Dijkstra, A Short Introduction to the Art of Programming. Report EWD 316, Technische Hogeschool, Eindhoven, The Netherlands, August 1971.

9. C. A. R. Hoare, Proof of a program: FIND. Comm. of the ACM, Vol. 14, No. 1 (January 1971), pp 39-45.

10. W. Wulf and M. Shaw, Global variable considered harmful. SIGPLAN Notices, Vol. 8, No. 2 (February 1973), pp 28-34.

11. C. A. R. Hoare, Hints on Programming Language Design. An invited address at the ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages, October 1973.

12. C. A. R. Hoare, Procedures and parameters: an axiomatic approach. Proceedings of the Symposium on the Semantics of Algorithmic Languages (E. Engeler, Ed.), Springer-Verlag, Berlin-Heidleberg-New York, 1971.

13. E. W. Dijkstra, 1972 ACM Turing Award Lecture: The humble programmer. Comm. of the ACM, Vol. 15, No. 10 (October 1972), pp 859-866.

14. E. I. Organick, Computer System Organization, The B5700/B6700 Series. Academic Press, New York and London, 1973.

15. H. Richards, Jr. and R. J. Zingg, The logical structure of the memory resource in the SYMBOL-2R computer. Proceedings of a Symposium on High-Level Language Computer Architecture, SIGPLAN Notices, Vol. 8, No. 11 (November 1973), pp 1-10.

16. J. W. Anderberg and C. L. Smith, High-level language translation in Symbol-2R. Proceedings of a Symposium on High-Level Language Computer Architecture, SIGPLAN Notices, Vol. 8, No. 11 (November 1973), pp 11-19.

17. P. C. Hutchison and K. Ethington, Program execution in the SYMBOL-2R Computer. Proceedings of a Symposium on High-Level Language Computer Architecture, SIGPLAN Notices, Vol. 8, No. 11 (November 1973), pp 20-26.

18. H. Richards, Jr. and C. Wright, Jr. Introduction to the SYMBOL-2R Programming language. Proceedings of a Symposium on High-Level Language Computer Architecture, SIGPLAN Notices, Vol. 8, No. 11 (November 1973), pp 27-33.

19. A. Hassitt, J. W. Lageschulte, and L. E. Lyon, Implementation of a high level language machine. Comm. of the ACM, Vol. 16, No. 4 (April 1973), pp 199-212.

20. L. P. Deutsch, A Lisp machine with very compact programs. Proceedings of the Third International Joint Conference on Artificial Intelligence, Stanford University, Stanford, Calif., August 1973.

21. B. H. Liskov, The design of the Venus operating system. Comm. of the ACM, Vol. 15, No. 3 (1972), pp 144-149.

22. J. Aiello, Investigation of Whether Existing Languages Can Support Data Representation for Structured Programming. S.M. Thesis, Department of Electrical Engineering, MIT, Cambridge, Mass., forthcoming.

23. N. Wirth, The programming language Pascal. Acta Informatica, Vol. 1, No. 1 (May 1971), pp 35-63.

24. S. N. Zilles, Data Algebra: A Specification Technique for Data Structures. Ph.D Thesis, Department of Electrical Engineering, MIT, Cambridge, Mass., forthcoming.

25. A. E. Fischer and M. J. Fischer, Mode modules as representations of domains. ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages (1973), pp 139-143.

26. F. J. Corbato and J. H. Saltzer, Multics -- the first seven years. AFIPS Conference Proceedings, Vol. 40, 1972, pp 571-583.

27. J. B. Dennis, Segmentation and the design of multiprogrammed computer systems. J. of the ACM. Vol. 12, No. 4 (October 1965), pp 589-602.

28. J. B. Dennis, Modularity. Advanced Course on Software Engineering, Lecture Notes in Economics and Mathematical Systems, Springer-Verlag, 1973, pp 128-182.

29. D. H. Vanderbilt, Controlled Information Sharing in a Computer Utility. Report TR-67, Project MAC, MIT, Cambridge, Mass., October 1969.

30. D. A. Henderson, Jr., The Binding Model: A Semantic Base for Modular Programming. Ph.D Thesis, Department of Electrical Engineering, MIT, Cambridge, Mass., forthcoming.

31. J. B. Dennis, Programming generality, parallelism and computer architecture. Information Processing 68, North-Holland Publishing Co., Amsterdam, 1969, pp 484-492.

32. J. B. Dennis, On the design and specification of a common base language. Proceedings of the Symposium on Computers and Automata, Polytechnic Press of the Polytechnic Institute of Brooklyn, N. Y., 1971, pp 47-74.

33. S. N. Amerasinghe, The Handling of Procedure Variables in a Base Language. S. M. Thesis, Department of Electrical Engineering, MIT, Cambridge, Mass. 1972.

34. J. B. Johnston, The contour model of block structured processes. Proceedings of a Symposium on Data Structures in Programming Languages, SIGPLAN Notices, Vol. 6, No. 2, ACM (February 1971), pp 55-82.

35. D. J. Ellis, Semantics of Data Structures and References. S. M. Thesis, Department of Electrical Engineering, MIT, Cambridge, Mass., forthcoming.

36. I. T. Hawryszkiewycz, Semantics of Data Base Systems. Ph.D Thesis, Department of Electrical Engineering, MIT, Cambridge, Mass., September 1973.

37. J. B. Fosseen, Representation of Algorithms by Maximally Parallel Schemata. S.M. Thesis, Department of Electrical Engineering, MIT, Cambridge, Mass., June 1972.

38. J. B. Dennis, Coroutines and parallel computation. Proceedings of the Fifth Annual Princeton Conference on Information Sciences and Systems, March 1971.

39. J. B. Dennis, A high-speed computer technique for the transportation problem. J. of the ACM, April 1958.

40. J. B. Dennis, A multi-user computer facility for education and research. Comm. of the ACM, Vol. 7, No. 9 (September 1964), pp 521-529.

41. C. A. R. Hoare and N. Wirth, An Axiomatic Definition of the Programming Language Pascal. Eidgenossische Technische Hochschule, Zurich, Berichte Der Fachgruppe Computer-Wissenschaften.

42. M. S. Paterson and C. E. Hewitt, Comparative schematology. Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, ACM, New York 1970, pp 119-127.

43. J. P. Linderman, Productivity in Parallel Computation Schemata. Report TR-111, Project MAC, MIT, Cambridge, Mass., December 1973.

44. J. B. Dennis and J. B. Fosseen, Introduction to Data Flow Schemas. Submitted for publication.

45. V. R. Pratt, Top down operator precedence. <u>ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages</u> (1973), pp 41-51.

46. J. H. Griesmer and R. D. Jenks, SCRATCHPAD/1 -- An interactive facility for symbolic mathematics. <u>Proc. Second ACM Symposium on Symbolic and Algebraic Manipulation</u>, 1971.